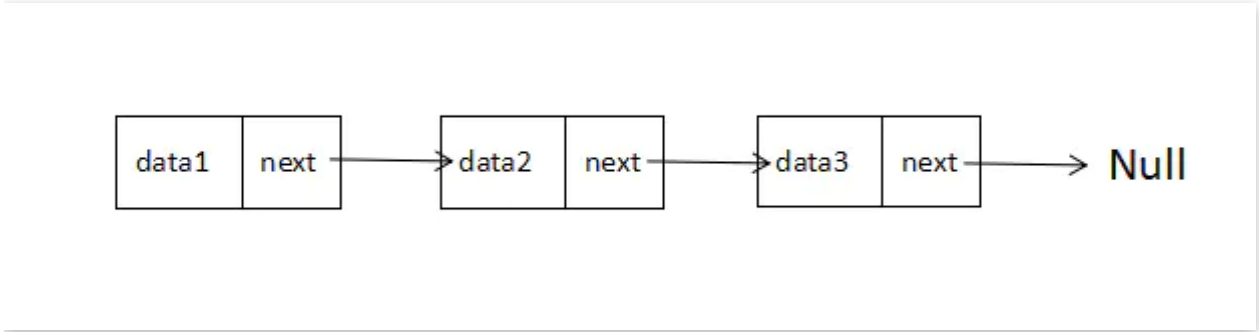
链表的定义

首先，要实现链表，我们先搞懂一些链表的基本东西，因为这很重要！

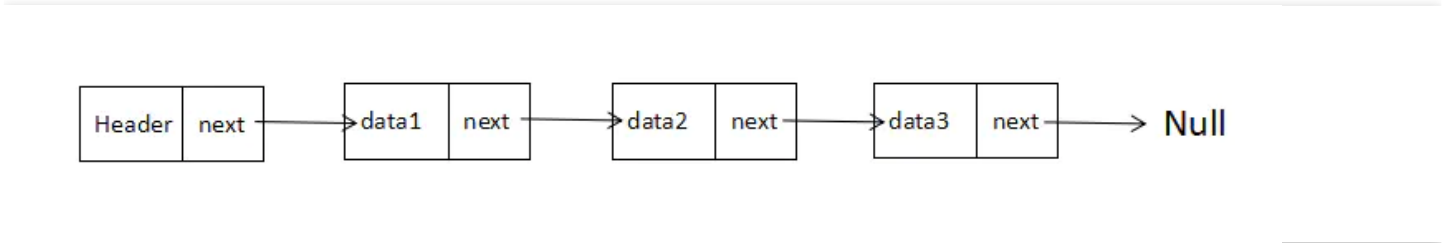
链表是一组节点组成的集合，每个节点都使用一个对象的引用来指向它的后一个节点。指向另一节点的引用讲做链。下面我画了一个简单的链接结构图，方便大家理解。



链表结构图

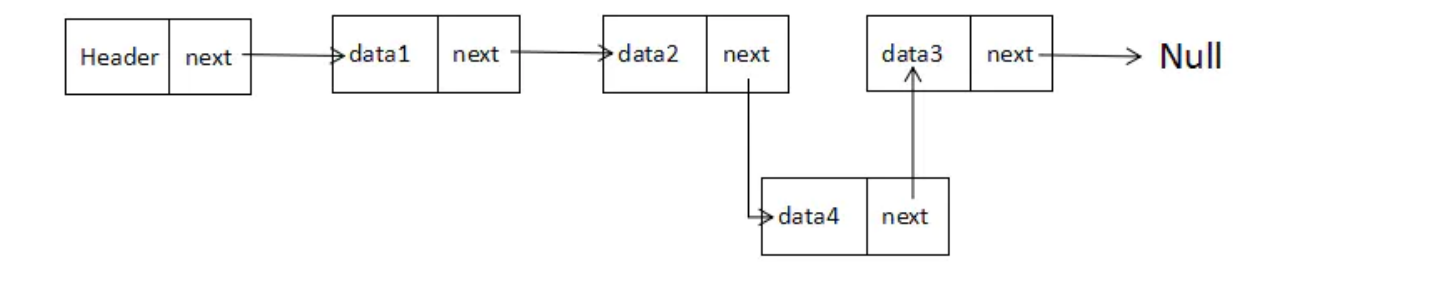
其中，data中保存着数据，next保存着下一个链表的引用。上图中，我们说 data2 跟在 data1 后面，而不是说 data2 是链表中的第二个元素。上图，值得注意的是，我们将链表的尾元素指向了 null 节点，表示链接结束的位置。

由于链表的起始点的确定比较麻烦，因此很多链表的实现都会在链表的最前面添加一个特殊的节点，称为 **头节点**，表示链表的头部。进过改造，链表就成了如下的样子：



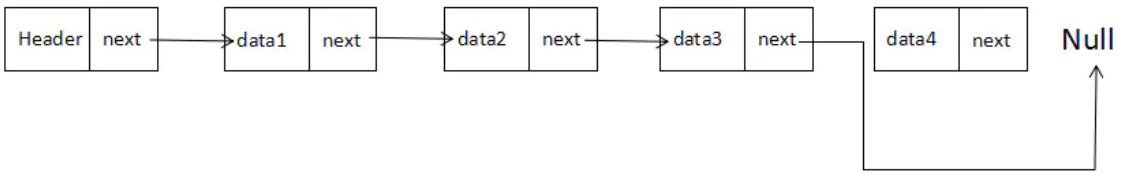
有头节点的链表

向链表中**插入一个节点**的效率很高，需要修改它前面的节点(前驱)，使其指向新加入的节点，而将新节点指向原来前驱节点指向的节点即可。下面我将用图片演示如何在 data2 节点 后面插入 data4 节点。



插入节点

同样，从链表中删除一个节点，也很简单。只需将待删节点的前驱节点指向待删节点的，同时将待删节点指向null，那么节点就删除成功了。下面我们用图片演示如何从链表中删除 data4 节点。



删除节点

链表的设计

我们设计链表包含两个类，一个是 Node 类用来表示节点，另一个事 LinkedList 类提供插入节点、删除节点等一些操作。

Node类

Node类包含连个属性： element 用来保存节点上的数据，next 用来保存指向下一个节点的链接，具体实现如下：

```
//节点
function Node(element) {
    this.element = element;    //当前节点的元素
    this.next = null;          //下一个节点链接
}
```

LinkedList类

LinkedList类提供了对链表进行操作的方法，包括插入删除节点，查找给定的值等。值得注意的是，它只有一个

属性，那就是使用一个 Node 对象来保存该链表的头节点。

它的构造函数的实现如下：

```
//链表类
function LList () {
    this.head = new Node( 'head' );      //头节点
```

```
    this.find = find;                   //查找节点
    this.insert = insert;               //插入节点
    this.remove = remove;               //删除节点
    this.findPrev = findPrev;           //查找前一个节点
    this.display = display;             //显示链表
}
```

head节点的next属性初始化为 null ，当有新元素插入时，next会指向新的元素。
接下来，我们来看看具体方法的实现。

insert：向链表插入一个节点

我们先分析分析insert方法，想要插入一个节点，我们必须明确要在哪个节点的前面或后面插入。我们先来看看，如何在一个已知节点的后面插入一个节点。
在一个已知节点后插入新节点，我们首先得找到该节点，为此，我们需要一个 find 方法用来遍历链表，查找给定的数据。如果找到，该方法就返回保存该数据的节点。那么，我们先实现 find 方法。

find：查找给定节点

```
//查找给定节点

function find ( item ) {
    var currNode = this.head;
    while ( currNode.element != item ){
        currNode = currNode.next;
    }
    return currNode;
}
```

find 方法同时展示了如何在链表上移动。首先，创建一个新节点，将链表的头节点赋给这个新创建的节点，然后在链表上循环，如果当前节点的 element 属性和我们要找的信息不符，就将当前节点移动到下一个节点，如果查找成功，该方法返回包含该数据的节点；否则，就会返回null。
一旦找到了节点，我们就可以将新的节点插入到链表中了，将新节点的 next 属性设置为后面节点的 next 属性对应的值，然后设置后面节点的 next 属性指向新的节点，具体实现如下：

```
//插入节点
function insert ( newElement , item ) {
```

```
    var newNode = new Node( newElement );
    var currNode = this.find( item );
    newNode.next = currNode.next;
    currNode.next = newNode;
}
```

现在我们可以测试我们的链表了。等等，我们先来定义一个 display 方法显示链表的元素，不然我们怎么知道对不对呢?

display：显示链表

```
//显示链表元素

function display () {
    var currNode = this.head;
    while ( !(currNode.next == null) ){
        console.log( currNode.next.element );
        currNode = currNode.next;
    }
}
```

实现原理同上，将头节点赋给一个新的变量，然后循环链表，直到当前节点的 next 属性为 null 时停止循环，我们循环过程中将每个节点的数据打印出来就好了。

```
var fruits = new LList();

fruits.insert('Apple' , 'head');
fruits.insert('Banana' , 'Apple');
fruits.insert('Pear' , 'Banana');

console.log(fruits.display());       // Apple
                                     // Banana
                                     // Pear
```

remove：从链表中删除一个节点

从链表中删除节点时，我们先要找个待删除节点的前一个节点，找到后，我们修改它的 next 属性，使其不在指向待删除的节点，而是待删除节点的下一个节点。那么，我们就得需要定义一个 findPrevious 方法遍历链表，检查每一个节点的下一个节点是否存储待删除的数据。如果找到，返回该节点，这样就可以修改它的 next 属性了。 findPrevious 的实现如下：

```
//查找带删除节点的前一个节点

function findPrev( item ) {
    var currNode = this.head;
    while ( !( currNode.next == null) && ( currNode.next.element != item )){
        currNode = currNode.next;
    }
    return currNode;
}
```

这样，remove 方法的实现也就迎刃而解了

```
//删除节点

function remove ( item ) {
    var prevNode = this.findPrev( item );
    if( !( prevNode.next == null ) ){
        prevNode.next = prevNode.next.next;
    }
}
```

我们接着写一段测试程序，测试一下 remove 方法：

```
// 接着上面的代码，我们再添加一个水果

fruits.insert('Grape' , 'Pear');
console.log(fruits.display());      // Apple
```

```
                                    // Banana
                                    // Pear
                                    // Grape

// 我们把香蕉吃掉

fruits.remove('Banana');
console.log(fruits.display());      // Apple
                                    // Pear
                                    // Grape
```

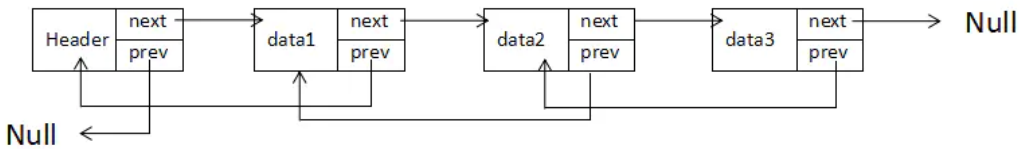
Great！成功了，现在你已经可以实现一个基本的单向链表了。

```
<script>
    // 节点
    function Node(element) {
        this.element = element;
        this.next = null;
    };
    // 链表
    function List() {
        this.head = new Node('head');     //头节点
        this.find = find;                 //查找节点
        this.insert = insert;             //插入节点
        this.remove = remove;             //删除节点
        this.findPrev = findPrev;         //查找前一个节点
        this.display = display;           //显示链表
    }

    // find: 查找给定节点
    function find(item) {
        let currentNode = this.head;
        while (currentNode.element != item) {
            currentNode = currentNode.next;
        }
        return currentNode;
    };

    // 插入节点
    function insert(newElement, item) {
        let newNode = new Node(newElement);
        let currentNode = this.find(item);
        newNode.next = currentNode.next;
        currentNode.next = newNode;
    };

    // 显示链表
    function display() {
        let currentNode = this.head;
        while (!(currentNode.next == null)) {
            console.log(currentNode.next.element);
            currentNode = currentNode.next;
        }
    };
    // eg:
```

```
    var fruits = new List();

    fruits.insert('Apple', 'head');
    fruits.insert('Banana', 'Apple');
    fruits.insert('Pear', 'Banana');

    console.log(fruits.display());

    // remove: 从链表中删除一个节点
    // 查找带删除节点的前一个节点
    function findPrev(item) {
        let currentNode = this.head;
        while (!(currentNode.next == null) && (currentNode.next.element != item)) {
            currentNode = currentNode.next;
        }
        return currentNode;
    };
    //删除节点
    function remove(item) {
        let prevNode = this.findPrev(item);
        if (!(prevNode.next == null)) {
            prevNode.next = prevNode.next.next;
        }
    };
    fruits.insert('Grape', 'Pear');
    console.log(fruits.display());// Apple// Banana// Pear// Grape

    fruits.remove('Banana');
    console.log(fruits.display()); // Apple
                                   // Pear
                                   // Grape
</script>
```

双向链表

尽管从链表的头节点遍历链表很简单，但是反过来，从后向前遍历却不容易。我们可以通过给Node类增加一个previous属性，让其指向前驱节点的链接，这样就形成了双向链表，如下图：



双向链表

此时，向链表插入一个节点就要更改节点的前驱和后继了，但是删除节点的效率提高了，不再需要寻找待删除节点的前驱节点了。

双向链表的实现

要实现双向链表，首先需要给 Node 类增加一个 previous 属性：

```
//节点类

function Node(element) {
    this.element = element;   //当前节点的元素
    this.next = null;         //下一个节点链接
    this.previous = null;     //上一个节点链接
}
```

双向链表的 insert 方法与单链表相似，但需要设置新节点的 previous 属性，使其指向该节点的前驱，定义如下：

```
//插入节点
function insert ( newElement , item ) {
    var newNode = new Node( newElement );
    var currNode = this.find( item );
    newNode.next = currNode.next;
    newNode.previous = currNode;
    currNode.next = newNode;
}
```

双向链表的删除 remove 方法比单链表效率高，不需要查找前驱节点，只要找出待删除节点，然后将该节点的前驱 next 属性指向待删除节点的后继，设置该节点后继 previous 属性，指向待删除节点的前驱即可。定义如下：

```
//删除节点

function remove ( item ) {
    var currNode = this.find ( item );
    if( !( currNode.next == null ) ){
        currNode.previous.next = currNode.next;
        currNode.next.previous = currNode.previous;
        currNode.next = null;
        currNode.previous = null;
    }
```

```
}
```

还有一些反向显示链表 dispReverse，查找链表最后一个元素 findLast 等方法，相信你已经有了思路，这里我给出一个基本双向链表的完成代码，供大家参考。

```
 //节点

function Node(element) {
    this.element = element;   //当前节点的元素
    this.next = null;         //下一个节点链接
    this.previous = null;         //上一个节点链接
}

//链表类

function LList () {
    this.head = new Node( 'head' );
    this.find = find;
    this.findLast = findLast;
    this.insert = insert;
    this.remove = remove;
    this.display = display;
    this.dispReverse = dispReverse;
}

//查找元素

function find ( item ) {
    var currNode = this.head;
    while ( currNode.element != item ){
        currNode = currNode.next;
    }
    return currNode;
}

//查找链表中的最后一个元素

```

```
function findLast () {
    var currNode = this.head;
    while ( !( currNode.next == null )){
        currNode = currNode.next;
    }
    return currNode;
}


//插入节点

function insert ( newElement , item ) {
    var newNode = new Node( newElement );
    var currNode = this.find( item );
    newNode.next = currNode.next;
    newNode.previous = currNode;
    currNode.next = newNode;
}

//显示链表元素

function display () {
    var currNode = this.head;
    while ( !(currNode.next == null) ){
        console.debug( currNode.next.element );
        currNode = currNode.next;
    }
}

//反向显示链表元素

function dispReverse () {
    var currNode = this.findLast();
    while ( !( currNode.previous == null )){
        console.log( currNode.element );
        currNode = currNode.previous;
    }
}

//删除节点
```

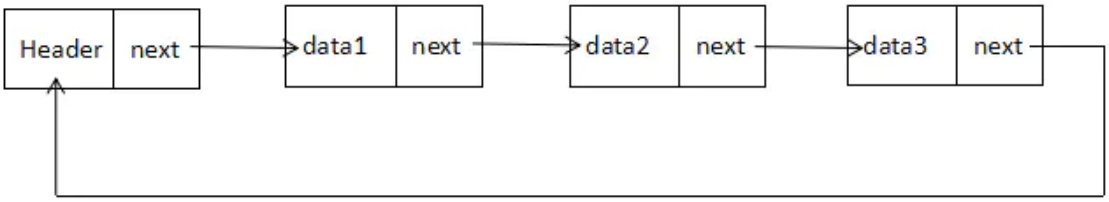
```
73
74 function remove ( item ) {
75     var currNode = this.find ( item );
76     if( !( currNode.next == null ) ){
77         currNode.previous.next = currNode.next;
78         currNode.next.previous = currNode.previous;
79         currNode.next = null;
80         currNode.previous = null;
81     }
82 }
83
84 var fruits = new LList();
85
86 fruits.insert('Apple' , 'head');
87 fruits.insert('Banana' , 'Apple');
88 fruits.insert('Pear' , 'Banana');
89 fruits.insert('Grape' , 'Pear');
90
91 console.log( fruits.display() );        // Apple
92                                         // Banana
93                                         // Pear
94                                         // Grape
95
96 console.log( fruits.dispReverse() );    // Grape
97                                         // Pear
98                                         // Banana
99                                         // Apple
```

循环链表

循环链表和单链表相似，节点类型都是一样，唯一的区别是，在创建循环链表的时候，让其头节点的 next 属性执行它本身，即

```
1 head.next = head;
```

这种行为会导致链表中每个节点的 next 属性都指向链表的头节点，换句话说，也就是链表的尾节点指向了头节点，形成了一个循环链表，如下图所示：



循环链表